Schedulability analysis by exhaustive state space construction: translating CCSL to transition-based Generalized Buchi Automata

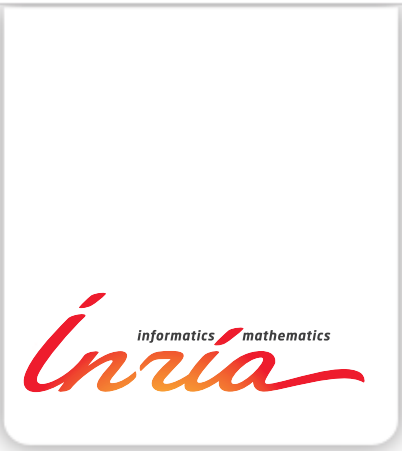
Ling Yin, Julien Deantoni, Frédéric Mallet, Robert de Simone

Schedulability analysis by exhaustive state space construction: translating CCSL to transition-based Generalized Büchi Automata

Ling Yin, Julien DeAntoni, Frédéric Mallet, Robert de Simone

RESEARCH REPORT
N° 8102
October 2012
Project-Teams Aoste

ISSN 0249-6399 ISRN INRIA/RR--8102--FR+ENG



Schedulability analysis by exhaustive state space construction: translating CCSL to transition-based Generalized Büchi Automata

Ling Yin, Julien DeAntoni, Frédéric Mallet, Robert de Simone

Project-Teams Aoste

Research Report n° 8102 — October 2012 — 22 pages

Abstract: In previous work we defined a language (CCSL) made to express real-time temporal scheduling constraints. It uses the notion of partially independent logical clocks (or time threads), of which seemingly physical discrete time is a special case, hence the name Clock Constraint Specification Language. Constraints can represent (asynchronous) causality and precedence relations, or (synchronous) simultaneity ones. A solution to a set of such constraints is called a schedule, as it brings back all the clocks down to a single, totally ordered discrete time (and thus allows to timing events a schedule slot). In the current paper we study the formal semantics of CCSL, by translation into specific automata recognizing infinite sequences of clock steps. We consider the appropriate acceptance criteria for omega-languages (transition-based generalized Büchi), and motivate this choice. We also study how the automata can be minimized, by removing useless states (which is more than the usual check for emptiness of the whole language in classical model-checking approaches). We feel our work builds a definite link between the scheduling and the formal semantic model-checking communities.

Key-words: CCSL; Scheduling; Automata

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Schedulability analysis by exhaustive state space construction: translating CCSL to transition-based Generalized Büchi Automata

Résumé : In previous work we defined a language (CCSL) made to express real-time temporal scheduling constraints. It uses the notion of partially independent logical clocks (or time threads), of which seemingly physical discrete time is a special case, hence the name Clock Constraint Specification Language. Constraints can represent (asynchronous) causality and precedence relations, or (synchronous) simultaneity ones. A solution to a set of such constraints is called a schedule, as it brings back all the clocks down to a single, totally ordered discrete time (and thus allows to timing events a schedule slot). In the current paper we study the formal semantics of CCSL, by translation into specific automata recognizing infinite sequences of clock steps. We consider the appropriate acceptance criteria for omega-languages (transition-based generalized Büchi), and motivate this choice. We also study how the automata can be minimized, by removing useless states (which is more than the usual check for emptiness of the whole language in classical model-checking approaches). We feel our work builds a definite link between the scheduling and the formal semantic model-checking communities.

Mots-clés : CCSL; Scheduling; Automata

1 Introduction

Previously we defined a formalism to express time relations between various events and operations of a given (discrete) system, named the Clock Constraint Specification Language [1]. CCSL is meant to capture some essential time and timing notions as found in various theories of timed concurrent processes, both from the synchronous and asynchronous sides. In this paper, we explore the use of CCSL for scheduling requirements modeling and schedulability analysis.

Historically, CCSL is based on a notion of "discrete clocks", which deserves (at least) a bit of explanation to avoid confusion. In our case a clock is not a physical device that provides regular physical time intervals (as a watch). It is instead an abstract measure unit, possibly user defined, by which durations and dates may be set or measured, as long as a sequence of occurrences ("ticks") on that clock can be observed or produced. In that sense a logical clock could also be named "logical time base", and largely corresponds in fact to just an abstract *event* (considered as a sequence of event occurrences), simply stressing more its temporal relevance and role in the design. Besides, logical clocks need not in any case be related to regular physical realities, but they can, and often do. To provide a simple example, a modern processor execution cycle, in these days of low-power and thermal budgets, is often quite irregular (it can even be paused on hold at times) with respect to physical time; nevertheless it is still a useful logical time unit when accounting for software complexity. Numerous other examples can be made up in the case of embedded systems, where contextual environment may produce meaningful events that punctuates the system behavior.

Based on clocks, CCSL introduces formal constructs, called clock constraints, both to define new clocks as functions of existing ones and also to specify relations between existing clocks as well. A significant list of primitive CCSL clock constraints will be presented in the paper, accounting for the language expressiveness and scheduling model description.

The logical clocks and clock constraints give CCSL the power to express most of the tasks/events models in (both real-time and non real-time) scheduling theories. A set of clocks and constraints among them can model the scheduling requirements (including functional requirements, platform features, resource limitations, extra timing requirements and etc), and will form a CCSL specification of the system. Solving the specification constraints amounts to computing a *schedule* of the system.

Under CCSL context, a schedule is a possibly infinite sequence of steps; for each step (index), several clocks (or maybe one) are assigned to tick simultaneously at that step. Then a schedule can be regarded as an infinite words on powerset of clocks. The general scheduling issues, like finding valid schedules or finding if a system under specific scheduling requirements is schedulable, can be related to formal language and automata theory on infinite words. Modeling scheduling concerns in CCSL allows the use of model checking technique and formal analysis to solve issues taken from the scheduling theory.

We do schedulability analysis by translating a CCSL specification into a transition-based Generalized Büchi automata. Proper acceptance criteria is defined to recognize valid schedules. Instead of finding an individual optimal schedule under a specific consideration, the automata representation represents all schedules that satisfy the requirements, and can be used as a basis for further analysis. The schedulability of a specification is equivalent to checking the emptiness of recognizable language of a corresponding automata. We also provide an algorithm to construct a trim canonical version for a given transition-based generalized automata. The trim automata can be used to guide later on scheduling.

The paper is constructed as follows: We introduce CCSL informally in section 2. Then we discuss its relation with existing scheduling practices and theories and the scheduling issues we want to deal with in section 3. In section 4 we show the translation from CCSL to transition-based

Generalized Büchi automata. Automata based solutions to the scheduling issues are provided in section 5. Then we conclude this paper, discuss related work and our future plans in section 6.

2 An introduction to CCSL

Tick and clock ordering relations

As already mentioned, clock ticks amount to event occurrences. Two event occurrences e and e' may temporally be: *simultaneous/coincident*, denoted as $e \equiv e'$, meaning that the two occurrences will occur in the same time step; or one may *precede* the other, denoted as $e \prec e'$, introducing a temporal ordering between them. We also note $e\#e'$ for *not* $(e \equiv e')$, and $e \preceq e'$ for $(e \prec e'$ *or* $e \equiv e')$. Note that causality intentions in design will in general allow relation $\preceq$, when instantaneous/combinatorial causality is also legible.

At the clock level (again, a clock being a sequence of ticks/instants), we find that two natural ordering relations extend the former. First, based on synchronous notions, a *subClock* relation can be defined: we note $c \sqsubset c'$ iff $\forall i, j : c_i \preceq c_j \ \exists k, l : (c_i \equiv c'_k) \wedge (c_j \equiv c'_l) \wedge (c'_k \preceq c'_l)$, c_i being the i^{th} tick of clock c; this definition means that every tick of c has to be made on a tick of c', but not always the other way around, so there are potentially more ticking instants in c'. Moreover, this relation is order preserving. Second, based on asynchronous notions, *faster than* relation can be defined: we note $c \prec c'$ iff $\forall i$, $c_i \prec c'_i$; similarly, $c \preceq c'$ iff $\forall i$, $c_i \preceq c'_i$ (strictly vs weakly faster).

The *subClock* relation will in practice be used to define new clocks from existing ones, with a subsampling mechanism. Often, the latter will be a regular pattern, so that periodic subsystems, or even periodically patterned one, will fall into this design style. Also, when original specifications are themselves not synchronously patterned, the *results* of scheduling methods are often computed of that nature, so that subclock relations with regular patterns will also be a mean to describe (full or partial solutions to) the system ordering; having a way to deal with partial solutions inside the syntax of the specification itself is useful, as it allows solving approaches by incremental time refinements.

The *faster than* relations can a priori lead to unbounded drifts between clocks (to ensure that the second clock does not tick too soon, we need to monitor how many ticks the first one has performed "in advance", just so it does not drop below 0, and this requires an integer in simulation). In frequent cases the specification will provide an interval between the clocks to bound drifts.

So far we have implicitly assumed that clocks were infinite sequences of ticks. Often, for technical reasons, one has to assume they can also be finite. So we will let $|c|$ denote the length of c, with the convention that $|c| = \infty$ when the clock is infinite. The previous definitions of clock ordering have to be adjusted so that they only apply where ticks are actually defined. This introduces no real change for the subClock relation, but it should be noted that, in the case of the faster than relation, c'_i may occur without a prior c_i, if $i > |c|$. This defeats the notion of causality between c and c' being encoded as a time ordering relation. To avoid such problem, when later constructs represent causality as such a time ordering, they will always rely on the fact that clock lengths will carefully match. Later, in dynamic operational semantic definitions, we shall use the predicate $Term(c)$ to mean that the proper termination of clock c, that is it has ticked up to its limit (whose value is generally dynamically reached).

2.1 Primitive CCSL clock functions

We now provide a set of primitive functions to build new clocks out of others. They are mostly of synchronous nature (subClock constructed).

The main function, `filteredBy`, will use an explicit mask to decide on each new tick (of the superclock) whether it should be preserved or discarded in the subclock definition. Such a mask consists of a binary word (with "1" meaning *keep* and "0" *discard*) of length at least equal to the length of the superclock. In practice, for syntactic reasons, the mask will have to be an ultimately periodic pattern, consisting of some initialization prefix part, and a steady/repeated part.
Simpler forms can be extracted using suitable masks, like `delay_n`, with n a constant number of ticks from the superclock; it simply discards the n first ticks of its input clock. Similarly `await_n` only ticks once, on the original n^{th} tick of its input clock, or `until_n`, which selects only the first n ticks of its input clock.
The `upto` operator takes two input clock arguments, and ticks with the first clock as long as the second one has not started.
The `concat` sequential composition also takes two input clocks. It will tick with the first until it has reached its length limit, then starts ticking as the second (when the first is terminated). Of course in practice this will require that checking for the first clock lifespan (or its length) is feasible, at least at run time. In simulation it can also mean that the property can be enforced by the simulation policy, but then it has to be checked that it is valid in the future.
`Union` (and `intersect`) functions work directly at the level of steps, and create new clocks that tick when at least one (or both) input clocks do.
The `sampledOn` function is a mix of synchronous and asynchronous spirit. It takes two input clocks and ticks synchronously with the second, but only in case the first input clock ticked since the previous tick of the second. So, its role is essentially to record whether there was a tick of the first clock since "last time".
`Inf` (and `sup`) functions also take two clocks as input, and create new clocks that is faster (or slower) than both of the input clocks. The i^{th} tick of the new defined clock is coincident with the earlier (or later) of the i^{th} tick of input clocks.

2.2 Primitive CCSL clock relations

Clock relations are provided to specify relations between two existing clocks. Both synchronous and asynchronous aspects are considered.
The fundamental synchronous relation is the `subClock` discussed above, restricting that the subclock cannot tick if its superclock does not. It is one direction specified, and can be extended into two directions, that is the `equality` relation stating the equality between the two clocks, one ticks iff the other one ticks.
The `exclusion` relation is the opposite of equality. Two clocks are exclusive means that they never tick at the same step.
The basic asynchronous relations are the two versions of `faster than` discussed in last subsection, the `strictPre` $\prec$ and `causes` $\preceq$. For the strict one, the second clock cannot tick when the index difference is 0. While for the weak one, when their index different is 0, the second clock can tick if the first one ticks. They both limit the drift between the two clocks in the range of $[0,\infty]$.
`BoundedDiff_i_j` bounds an integer interval [i,j] to the drift of the two involved clocks, denoted as $i \leq left - right \leq j$. It can be viewed as an extension of faster than, not restricting which clock is faster, but stating their index difference cannot exceed the range.
The relation `alternate` indicates the alternate occurrences of instants of the two clocks, starting from the left one.

Considering clock termination: the `filteredBy` operator preserves termination, and may create a finite clock from an infinite one (if the mask is of finite length); The `delay_n, await_n`

and `upto` operators create finite clocks (unless the second clock of upto never starts); `concat` will be of real use only if its first clock argument terminates; finite `union` clock requires the two clock arguments to be finite, while `intersect` may produce a finite clock even both arguments are infinite; `sampledOn` is finite if one of its argument is; by convention `inf` provides infinite clocks as soon as one of their arguments is infinite, while `sup` provides finite one as soon as one argument is finite. A subclock of a finite clock is finite (while subclock of an infinite clock may be finite too); a clock in bounded drift of a finite one is finite (a clock with a finite `faster than` clock is finite as well).

3 The issue: schedulability analysis through formal translation to a proper operational semantic domain

We want to play the connection between the scheduling theory and the automata theory through CCSL, modeling the issues taken from scheduling theory by CCSL, and solving them by translating a CCSL specification into a proper automata with acceptance criteria recognizing specific infinite words. First we discuss the relation between CCSL and existing scheduling theories. Then we present the scheduling issues in CCSL context and provide the overview of our automata based solutions to the issues.

3.1 Relations between CCSL and existing scheduling theories

Real-time scheduling theories rely on task models that supposedly abstract real applications. Originally they were rather simple (e.g., independent periodic tasks only for Rate Monotonic Analysis). Always more sophisticated models now appear in the literature. They are all based on numerous distinct parameters, providing numerical constraint values for timing aspects (dispatch time, period, deadline, jitter drift,...). Tasks are considered as iterations of jobs (or jobs as instances of tasks). In our view, the successive timing values for characteristic feature of successive jobs can be seen each as a logical clock, and the time constraint relations between such clocks are usually expressed as simple equalities and bounded inequalities that fall well into the range of CCSL constructs descriptive power. As an example, the notion of *sporadicity* which states that tasks are invoked at random but with a minimum inter-invocation interval, can be handled by CCSL, as for instance, $c = (p$ `filteredBy` $(001)^\omega)$ `sampledOn` d, $0 \leq d - c \leq 1$ and $d \prec (p$ `wait_1`$)$, states at least three ticks of p between two ticks of d; the fact that p ticks are regular in physical time is ignored in CCSL modeling. Note that logical clock in CCSL can also be used to represent realities of different nature, more ad-hoc to the application, with added intuitive expressiveness. For instance, the exclusive access to a critical resource is modeled by `exclusion` between corresponding clocks, the size limit of a buffer (3 as an example) can be expressed by a special case of `boundedDiff` relation, making 0 as the left bound and buffer size as the right bound between *write* and *read* clocks, $0 \leq write - read \leq 3$. With such formal specification, CCSL is not able to say anything about the way to find efficient resulting schedule(s) for specific class of problems modeled that way (for instance, Earliest Deadline First plans to schedule first the job with least time remaining, this remaining time being dynamically computed due to past activities). Further comparisons of CCSL and usual real-time scheduling models would be out of the scope of the current paper. But it should still be mentioned that models closer to CCSL, as for instance AADL and AutoSar-related EAST-ADL models, are also now making their way to mainstream scheduling concerns.

Classical (non real-time) scheduling, on its side, provides generally models where the initial constraints are less on timing and more on dependencies (as for program instruction scheduling

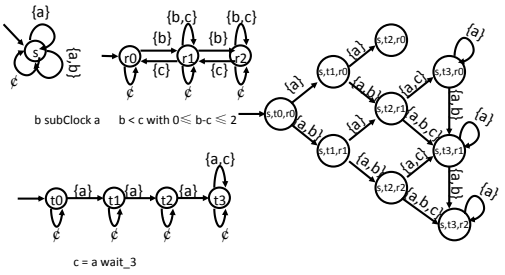


Figure 1: Execution of specification in Example 1

and software pipelining) or on exclusive resource allocation. But resulting schedules are almost always of *modulo* periodic nature, here again matching the CCSL expressiveness.

3.2 Automata based solution to CCSL scheduling issue

A CCSL specification formally describes expected requirements and represents a set of valid schedules, `i.e.` a set of valid sequences of steps; where for each steps (index), clocks are said to tick or not. Solving the specification constraints consists in assigning clock ticks to steps, and consequently to provide one schedule of the system.

The purpose of scheduling is to assign resources and time slots to tasks so that all tasks are accomplished under some constraints (e.g., fair share of resources or time, using as little time slots as possible, etc). In this sense, a valid schedule should be a sequence of steps where every clock ticks as far as it meant to (infinitely often or up to its proper termination). Specifications may become unschedulable because of contradictory constraints, or more subtly because it becomes unfeasible to pack a number of ticks in a given time interval to get a valid schedule.

We provide a typical example below, which will reveal a number of concerns involved in CCSL schedulability.

Example 1. For three clocks a, b and c, where b is a subclock of a, c is a `wait 3`, b is strictly `faster than` c and their index difference is bounded in [0,2]:

$$b \texttt{ subClock } a,\ c = a \texttt{ wait 3},\ b \prec c,\ 0 \leq b - c \leq 2$$

The automata like representation in Figure 1 is informally used here to give readers an intuitive view of the issue. Formal semantics is given later. The global behaviors of a specification (on the right in Figure 1) have to respect to all its constraints (on the left in Figure 1). One major problem is that during the construction of the schedules, some choices can lead to invalid schedules. For instance, what is expected from our example is: At least one tick of b should occur before the third tick of a, then c ticks together with the third tick of a and then terminates. Clock b terminates when the index difference between b and c reaches 2, and a ticks infinitely. However, none of the constraints forbids a to tick two times without one b. In state $< s, t_2, r_0 >$ it becomes unfeasible to fire any clock with respect to the conjunction of constraints: every clocks get halted (cannot tick anymore though it is not proper terminated). The run $< s, t_0, r_0 > \xrightarrow{\{a\}} < s, t_1, r_0 > \xrightarrow{\{a\}} < s, t_2, r_0 >$ consequently cannot produce a valid schedule.

To decide the schedulability of a specification and to recognize valid schedules from specification runs, we provide a state-based operational semantics for CCSL, ccAutomata. Such ccAutomata will represent all possible runs of a specification, and thus all valid schedules of its clock ticks. Recognizing valid schedules from runs can be achieved by defining acceptance criteria

for infinite words on ccAutomata. The schedulability issue, which demands that one such valid schedule exists, will thus amounts to the emptiness checking problem.

4 Formal translation of CCSL into (Generalized Transition Büchi) ccAutomata

We will provide the formal translation of CCSL clock constraints into appropriate state-based models using operational semantics. We will first describe how the "stateful" features of operators expand into states, providing the general skeletons for the expanded models; then we will consider the acceptance criteria issue, so that exactly valid schedules are recognized. In the next section we will show how the resulting models can be used to solve the schedulability issues.

Acceptance criteria for languages of infinite words have been amply studied in the past. This gave rise to recognition patterns such as Büchi (repeated states), certainly the most famous, but also Muller, Rabin, and Street types of acceptance criteria. The Büchi acceptance criterion also supports variants, such as transition Büchi (rather than state Büchi), or Generalized Büchi criteria (in which several sets of repeated states are provided and must be satisfied collectively).

We shall show how CCSL naturally leads to transition Büchi automata translation for individual constraints, and why state Büchi is less natural; we shall also see how the generalized version is needed when specification are composed of several distinct clocks (and there will be one recognition set per clock). It is known from the literature that both variants can be expanded at some cost into plain (state) Büchi automata, but we will consider how direct analysis may prove to be more efficient.

For homogeneity reasons we shall also see how termination states and finite clocks can be unified with infinite ones (by noticing infinitely the termination). We will also have to rely on stuttering transitions so that the construction of a global automaton from the various constraints (and their individual automaton form) can be considered as a synchronous product, even though some may idle in it. These can be seen as "classical tricks" in the model-checking community, but they are to be applied on models and theories meant to the (distinct) scheduling theory community.

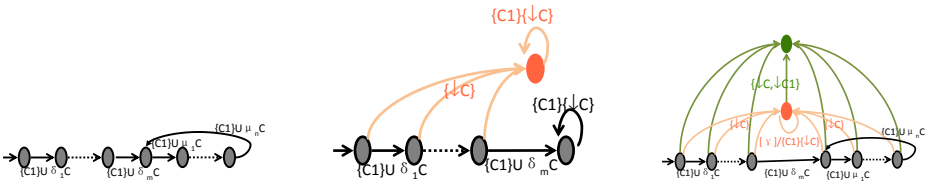
4.1 State-based operational semantics of CCSL—ccAutomata

Definition 3. A ccAutomata is a tuple $\mathcal{A} = (S, Clocks, Terms, I, T)$ where

- S is a nonempty set of states, and $I \subseteq S$ is the set of initial states.
- $Clocks$ is a finite nonempty set of clock names, which can be regarded as sorts of the automata.
- $Terms$ is a set of clock termination predicates $Terms = \{Term(c) | c \in Clocks\}$.
- $T \subseteq S \times 2^{Clocks} \times 2^{Terms} \times S$ defines the transition relation, each transition is labeled by a subset of Clocks that tick simultaneously in that step, and a subset of Terms indicating the proper terminations of corresponding clocks.

A transition $(s, C, Tm, s') \in T$ is also denoted as $s \xrightarrow{C,Tm}_T s'$ or simply $s \xrightarrow{C,Tm} s'$ if clear from context. A ccAutomata is *deterministic* iff $\forall s \in S, (s \xrightarrow{C,Tm} s_1 \wedge s \xrightarrow{C,Tm} s_2) \Rightarrow (s_1 = s_2)$.

A *path* of $\mathcal{A}$ is a sequence of transitions, either finite or infinite, $\rho = s_0 \xrightarrow{C_0,Tm_0} s_1 \xrightarrow{C_1,Tm_1} \cdots \xrightarrow{C_{k-1},Tm_{k-1}} s_k \cdots$. A *run* is a path from an initial state, and its *ticking trace* is the series of ticking labels: $C_0; C_1; \cdots C_k; \cdots$.



(a) $\oint(c) = \oint(c_1) = Inf$ (b) $\oint(c) = Fin, \oint(c_1) = Inf$ (c) $\oint(c) = \oint(c_1) = Fin$

Figure 2: $c = c_1$ `filteredBy` $u(v)^\omega$

4.1.1 ccAutomata encodings of CCSL clock constraints

One can easily build a ccAutomata for a clock constraint according to its semantics. If a clock terminates in one transition, then it is terminated in any future transition, i.e., if $s \xrightarrow{C,Tm} s', Term(c) \in Tm$, then $\forall t, s' \xrightarrow{C',Tm'} t, Term(c) \in Tm'$. Every state has a stutter transition $s \xrightarrow{\emptyset,Tm} s$ since no constraint is violated or changed if no clock ticks.

Function $\oint : Clocks \rightarrow \{Fin, Inf, Free\}$ is provided to user to specify that a clock is supposed to be finite ($\oint(c) = Fin$), infinite ($\oint(c) = Inf$), or either ($\oint(c) = Free$). It needs to be in consistent with the constraint semantics, for instance, for $c = c_1$ `await_3`, $\oint(c)$ must be Fin. The $\oint$ function can be used to simplify the ccAutomata encoding (for scheduling concerns). If we know a clock is specified to be infinite, the the transitions labeled by its termination can be removed since termination of that clock is not expected (transitions labeled by that will not lead to valid schedules). We will show how the $\oint$ is used to simplify the encoding: for each primitive clock constraint, we first show its ccAutomata encoding for the most natural case (usually the case that $\oint(c_1) = \oint(c_2) = Inf$), then add corresponding termination transitions and states for finite specified clocks. These transitions and states are colored for clear view, where "red" means "un-conclusive", some clocks have terminated while others haven't, and "green" means "terminal" that every clock has terminated.

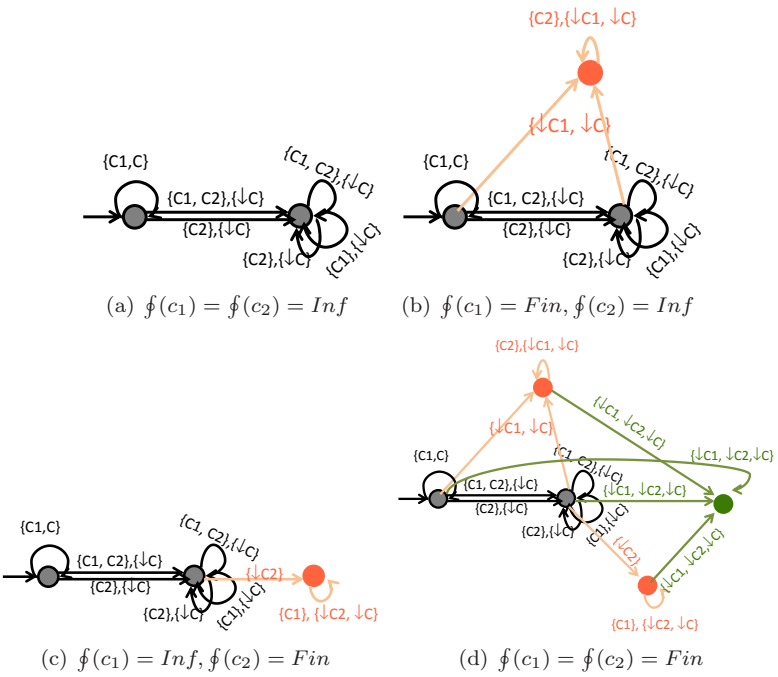
For simplicity, termination predicts are denoted by $\downarrow clockname$, and stutter transitions are ignored in the figures.

- `filteredBy, delay_n,await_n,until_n`
  Constraint $c = c_1$ `filteredBy` w builds a sub clock c of clock c_1 according to the binary word w. Usually w is written as $u(v)^\omega$, where u is the transient part and v the periodic part with length m, v is the periodic part with n bits, the power ω means v is repeated an unbounded number of times. When the bit of w is 1, the instant of c_1 is selected, otherwise discarded. $\oint(c_1) = Fin \Rightarrow \oint(c) = Fin$.

  The most natural case is $\oint(c) = \oint(c_1) = Inf$ shown in Figure 2(a), where $\delta_i c = \{c\}$ if $u_i = 1$, otherwise $\delta_i c = \emptyset$, similarly $\mu_i c = \{c\}$ when $v_i = 1$, $\mu_i c = \emptyset$ when $v_i = 0$, and $\gamma = 1$ if the next bit of w is 0.

  If c is specified to be finite ($\oint(c) = Fin, \oint(c_1) = Inf$), then v must be 0. Corresponding transitions and states are added, c_1 alone ticks infinitely after the termination of c, as Figure 2(b) shows.

  If c_1 is also specified to be finite ($\oint(c) = \oint(c_1) = Fin$), transitions labeled with $Term(c_1)$ are added. As Figure 2(c) shows, if c_1 terminates, c terminates directly. While c_1 can still tick until the bit of w is 1 after the termination of c.



(a) $\oint(c_1) = \oint(c_2) = Inf$ (b) $\oint(c_1) = Fin, \oint(c_2) = Inf$

(c) $\oint(c_1) = Inf, \oint(c_2) = Fin$ (d) $\oint(c_1) = \oint(c_2) = Fin$

Figure 3: ccAutomata encodings of $c = c_1$ `upto` c_2

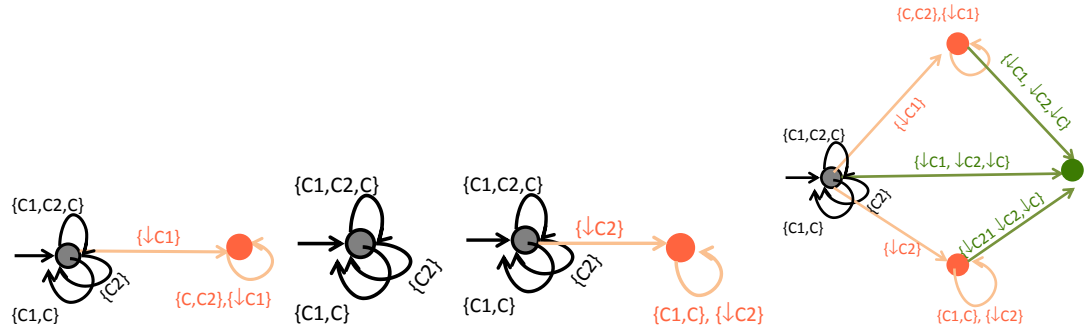
Constraints `delay_n,await_n,until_n` can be regarded as special cases of `filteredBy`, using $0^n(1)^\omega, 0^n1(0)^\omega$ and $1^n(0)^\omega$ as masks respectively.

- `upto`
  $c = c_1$ `upto` c_2 creates a finite clock unless c_2 never starts and c_1 is infinite. Clock c terminates properly when c_2 starts.

- `concat`
  Constraint $c = c_1$ `concat` c_2 relies on dynamic terminations, c behaves like c_1 until its termination and then behaves c_2. Usually c_1 should be finite, otherwise c is just equal to c_1.

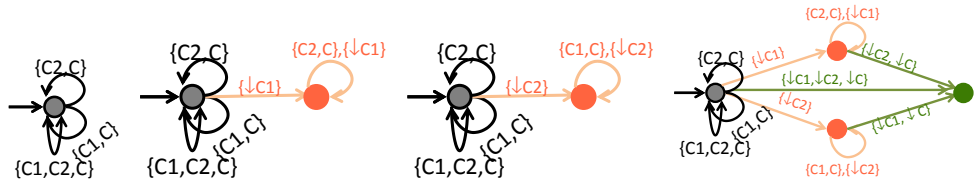
- `union,intersection`
  Constraint $c = c_1$ `union` c_2 creates a new clock c which ticks whenever c_1 or c_2 ticks. c is finite if and only if both c_1 and c_2 is finite. If only c_1 (resp. c_2) terminates, c can still tick together with c_2 (resp. c_1) until the termination of c_2 (resp. c_1). On the opposite, $c = c_1$ `intersection` c_2 creates c which ticks iff both of c_1 and c_2 tick. If one of c_1 or c_2 terminates, c terminates, and the other clock can tick till its termination; if c terminates, c_1 and c_2 can tick infinitely in the future but not together.

- `sampledOn`
  There are two versions of the sampling, either strict ($c = c_1$ `strictSamp` c_2), or non-strict ($c = c_1$ `nstrictSamp` c_2). In both versions, clock c_1 is considered as a trigger and c_2 is a time base, they do not affect each other, and clock c ticks in coincidence with the tick of c_2 immediately following a tick of the trigger c_1. Their difference can be clearly described by the following expressions:



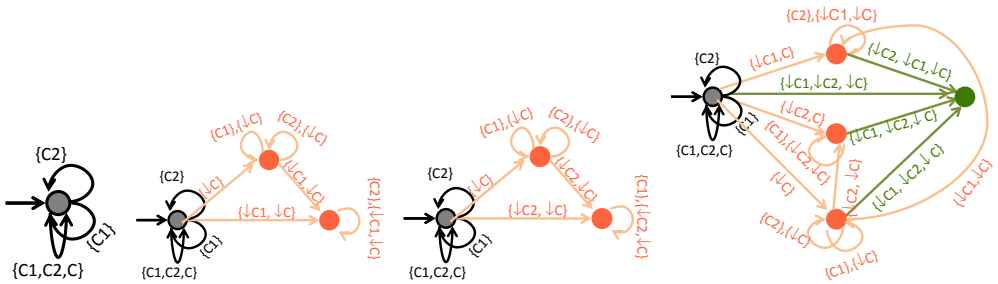
(a) $\oint(c_1) = Fin, \oint(c_2) = Inf$ (b) $\oint(c_1) = Fin, \oint(c_2) = Inf$ (c) $\oint(c_1) = Inf, \oint(c_2) = Fin$ (d) $\oint(c_1) = \oint(c_2) = Fin$

Figure 4: ccAutomata encodings of $c = c_1$ `concat` c_2



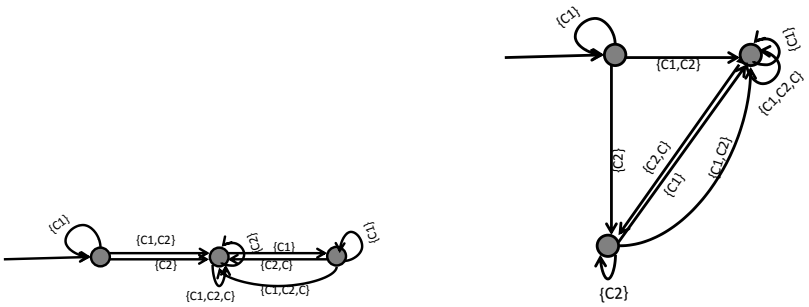
(a) $\oint(c_1) = Fin, \oint(c_2) = Inf$ (b) $\oint(c_1) = Fin, \oint(c_2) = Inf$ (c) $\oint(c_1) = Inf, \oint(c_2) = Fin$ (d) $\oint(c_1) = \oint(c_2) = Fin$

Figure 5: ccAutomata encodings of $c = c_1$ `union` c_2



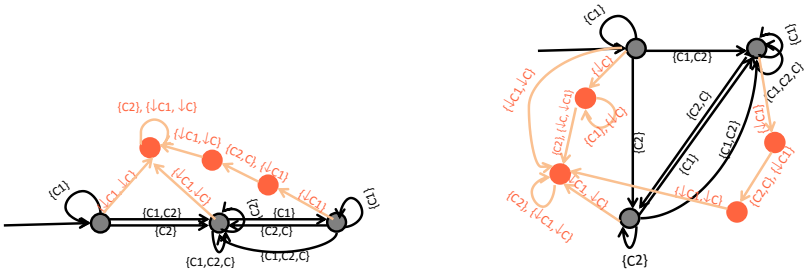
(a) $\oint(c_1) = Fin, \oint(c_2) = Inf$ (b) $\oint(c_1) = Fin, \oint(c_2) = Inf$ (c) $\oint(c_1) = Inf, \oint(c_2) = Fin$ (d) $\oint(c_1) = \oint(c_2) = Fin$

Figure 6: ccAutomata encodings of $c = c_1$ `intersect` c_2



(a) $c = c_1$ `nstrictSamp` c_2 (b) $c = c_1$ `strictSamp` c_2

Figure 7: ccAutomata encoding of sampledOn($\oint(c_1) = \oint(c_2) = Inf$)



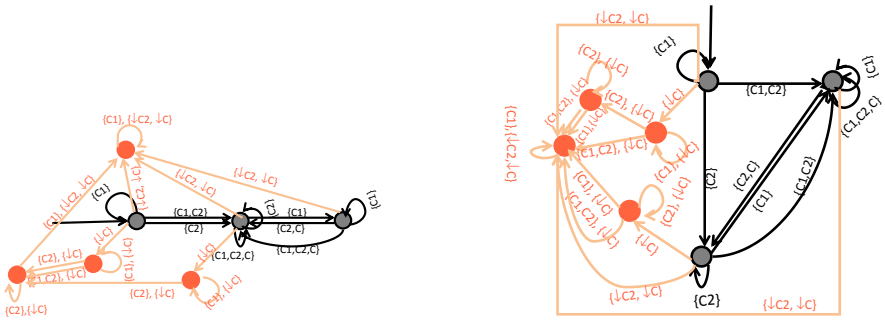
(a) $c = c_1$ `nstrictSamp` c_2 (b) $c = c_1$ `strictSamp` c_2

Figure 8: ccAutomata encoding of sampledOn($\oint(c_1) = Fin, \oint(c_2) = Inf$)

$$\text{strict: } \forall i, \exists j, k, c_2[j-1] \preceq c_1[i] \prec c_2[j], c[k] \equiv c_2[j]$$
$$\text{weak: } \forall i, \exists j, k, c_2[j-1] \prec c_1[i] \preceq c_2[j], c[k] \equiv c_2[j]$$

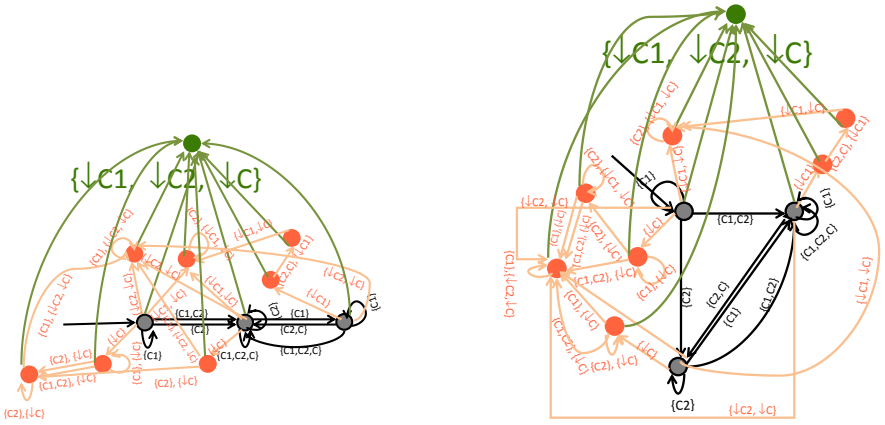
Please check the difference reflected in ccAutomata encodings: Figure 4.1.1 for $\oint(c_1) = \oint(c_2) = Inf$ case; Figure 4.1.1 for $\oint(c_1) = Fin, \oint(c_2) = Inf$ case; Figure 4.1.1 for $\oint(c_1) = Inf, \oint(c_2) = Fin$ case; and Figure 4.1.1 for $\oint(c_1) = \oint(c_2) = Fin$ case.

- `inf, sup`
  Constraint $c = c_1$ `sup` c_2 defines a new clock c which is slower than both c_1 and c_2. The k^{th} tick of c is coincident with the later of the i^{th} tick of c_1 and c_2. The index difference δ between c_1 and c_2 is recorded to determine if c should tick together with which clock, $\delta = i$ at s_i. When $\delta = 0$, if c_1 (resp. c_2) terminates, then c terminates; if c terminates, then either only c_1 ticks or only c_2 ticks in the future. When $\delta \neq 0$, the slower one c_2 (resp. c_1) terminates iff c terminates; if c_1 (resp. c_2) terminates, then c can be alive till δ becomes 0, c_2 (resp. c_1) can still tick after the terminations of c and c_1 (resp. c_2). Constraint $c = c_1$ `inf` c_2 is the dual of $c = c_1$ `sup` c_2. The k^{th} tick of c is coincident with the faster of the i^{th} tick of c_1 and c_2. When $\delta = 0$, if c terminates, then both c_1 and c_2 terminates; if c_1 (resp. c_2) terminates, then c_2 (resp. c_1) and c ticks together till their terminations. When $\delta > 0$ (resp. $\delta \neq 0$): if the slower one terminates then c and the faster one tick together in the future up to their terminations; if c terminates, then the faster one terminates immediately, the slower one can tick till δ becomes 0; if the faster one terminates, the slower one ticks alone till δ becomes 0, then ticks together with c before terminate.



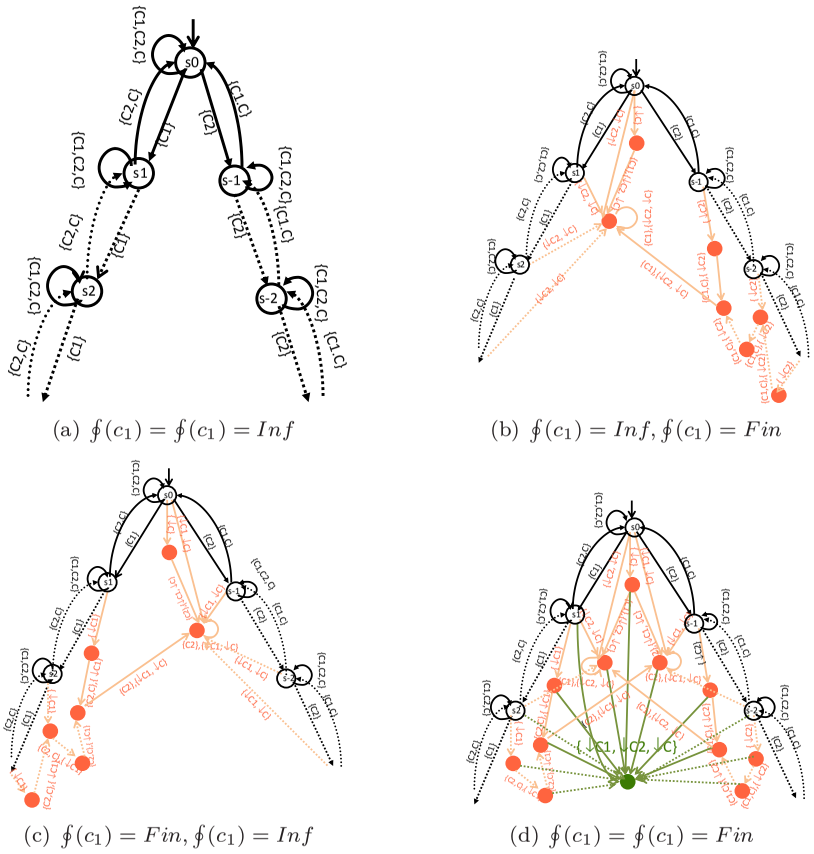
(a) $c = c_1$ `nstrictSamp` c_2 (b) $c = c_1$ `strictSamp` c_2

Figure 9: ccAutomata encoding of sampledOn($\oint(c_1) = Inf, \oint(c_2) = Fin$)



(a) $c = c_1$ `nstrictSamp` c_2 (b) $c = c_1$ `strictSamp` c_2

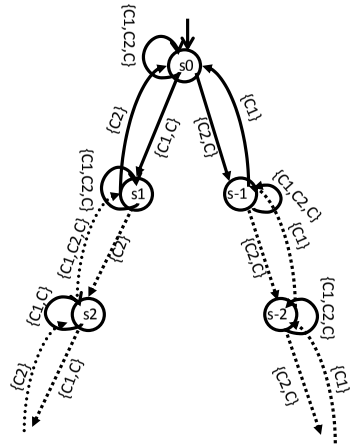
Figure 10: ccAutomata encoding of sampledOn($\oint(c_1) = \oint(c_2) = Fin$)



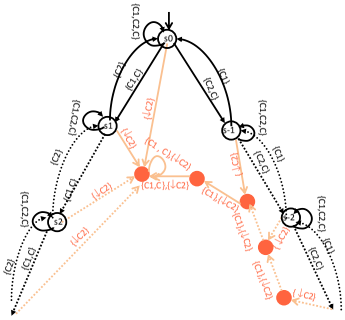
(a) $\oint(c_1) = \oint(c_1) = Inf$ (b) $\oint(c_1) = Inf, \oint(c_1) = Fin$

(c) $\oint(c_1) = Fin, \oint(c_1) = Inf$ (d) $\oint(c_1) = \oint(c_1) = Fin$

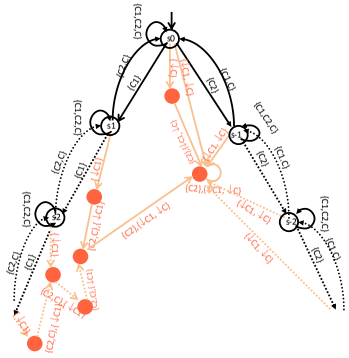
Figure 11: ccAutomata encoding of $c = c_1$ `sup` c_2



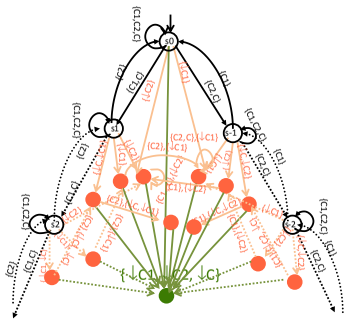
(a) $\oint(c_1) = \oint(c_1) = Inf$



(b) $\oint(c_1) = Inf, \oint(c_1) = Fin$

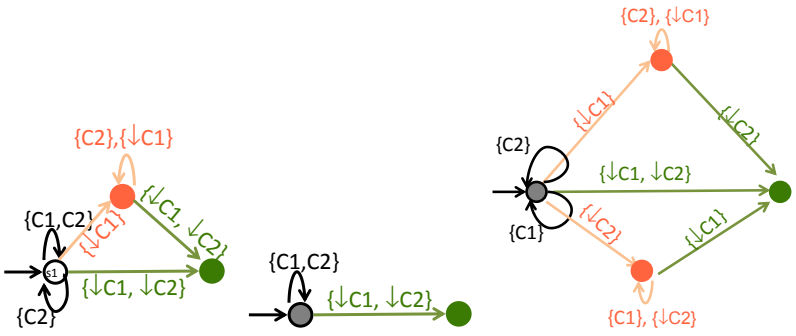


(c) $\oint(c_1) = Fin, \oint(c_1) = Inf$



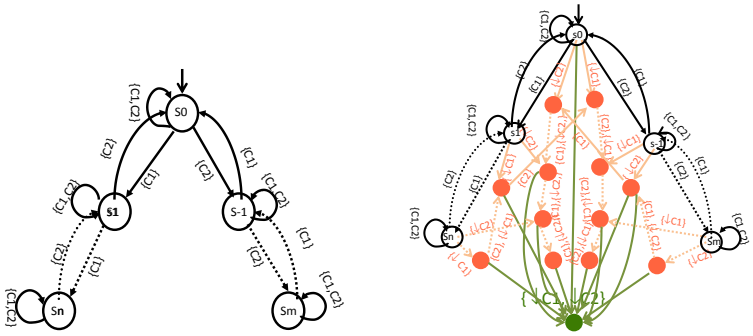
(d) $\oint(c_1) = \oint(c_1) = Fin$

Figure 12: ccAutomata encoding of $c = c_1$ `inf` c_2



(a) $c = c_1$ `subClock` c_2 (b) $c = c_1$ `equality` c_2 (c) $c = c_1$ `exclusion` c_2

Figure 13: ccAutomata encodings of `subClock, equality,exclusion`



(a) $\oint(c_1) = \oint(c_2) = Inf$ (b) $\oint(c_1) = \oint(c_2) = Fin$

Figure 14: ccAutomata encoding of `BoundedDiff`

- `subClock, equality,exclusion`
  c_1 `subClock` c_2, c_1 is a subclock of c_2, so either both of them tick, or only c_2 tick, or none of them tick. If c_2 terminates, c_1 also terminates. While after the termination of c_1, c_2 can still tick till its termination. If a clock is specified to be infinite, the ccAutomata is obtained by just removing the transitions labeled by its termination.

  c_1 `equality` c_2 means c_1 `subClock` c_2 and c_2 `subClock` c_1, as Figure 13(b) shows.

  c_1 `exclusion` c_2 specifies that they cannot tick together. Their terminations do not affect each other.

- `BoundedDiff, strictPre, causes`
  If $m \leq c_1 - c_2 \leq n$ $(i, j \in N)$, then $\oint(c_1) = \oint(c_2)$. Similar to `sup` and `inf`, the index difference $\delta = i$ at state s_i. The ccAutomata encoding for Inf case is shown in Figure 14(a). For Fin case, at s_i, if c_2 (resp. c_1) terminates, c_1 (resp. c_2) can at most $n - i$ times before termination.

  Constraint `causes` is a special case of `BoundedDiff` that $m = 0, n = \infty$. `strictPre` is the same as causes except that at s_0 the slower clock cannot tick.

- `alternate`
  c_1 `alternate` c_2 indicates the alternate occurrences of instants of c_1 and c_2. It implies

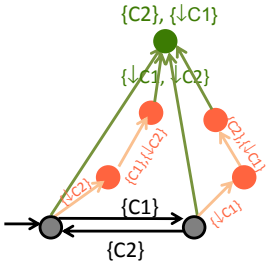


Figure 15: c_1 `alternate` c_2

$0 \leq c_1 - c_2 \leq 1$. Its ccAutomata encoding for Fin case is shown in Figure **??**. The Inf case is easy gotten by removing transitions labeled by terminations.

4.1.2 ccAutomata semantics of CCSL specifications—composition of individual ccAutomatas

A specification is a conjunction of constraints so its resulting behavior have to respect all its constraints. Transitions from two ccAutomatas can be composed only if there is no conflict; conflict here means that a common involved clock ticks in one transition while does not in the other, or it terminates in one transition while stays alive in the other.

Definition 2. Given a set of ccAutomatas, $\mathcal{A}_i = \{S_i, Clocks_i, Terms_i, I_i, T_i\}$, where $i \in \zeta = \{1, ..., n\}$, their *composition* is a proper synchronized product, $||\{\mathcal{A}_i\}_{i\in\zeta} = \{S, Clocks, Terms, I, T\}$, where

$$S = S_1 \times \cdots \times S_n; I = I_1 \times \cdots \times I_n;$$
$$Clocks = Clocks_1 \cup \cdots \cup Clocks_n; Terms = Terms_1 \cup \cdots \cup Terms_n;$$
$$\frac{\forall i, j \in \zeta(i \neq j), s_i \xrightarrow{C_i, Tm_i}_{T_i} s_i^{'}, s_j \xrightarrow{C_j, Tm_j}_{T_j} s_j^{'}, \forall c \in Clocks_i \cap Clocks_j, \quad c \in C_i \Leftrightarrow c \in C_j, Term(c) \in Tm_i \Leftrightarrow Term(c) \in Tm_j}{(s_1, ..., s_n) \xrightarrow{C_1 \cup \cdots \cup C_n, Tm_i \cup \cdots \cup Tm_n}_{T} (s_1^{'}, ..., s_n^{'})}$$

4.2 Acceptance criteria for valid schedules

A valid schedule is the ticking trace of a run, in which every clock ticks as far as it meant to. That is to say, every infinite specified clock ticks infinitely often and every finite specified clock ticks up to its proper termination.Thanks to the stutter transitions and the preservation of clock terminations, a clock ticks up to its proper termination can be noticed by the infinite visit of its termination predict. A run ends in a state s_f where every clock is terminated properly can always be extended to infinite by $s_f \xrightarrow{\emptyset, Terms} s_f$. Then acceptance criteria for infinite words can be used to recognize valid schedules.

We say path ρ *passes infinitely often* through a state (or transition in transition-based style) if there are infinitely many occurrences of it in ρ. The set of states (or transitions) which occur infinitely often in ρ is denoted by $Inf(\rho)$.

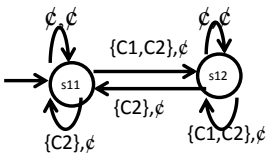


Figure 16: Expand state s_1 in Figure 13(a) into s_{11} and s_{12}

Acceptance criteria in Büchi recognizing pattern is a set of repeated states, i.e., a state-based Büchi ccAutomata (denoted as state BA for short) is a tuple $(\mathcal{A}, F)$, where $\mathcal{A}$ is a ccAutomata, $F \subseteq S$ is a finite set of accepting states. A run of $\mathcal{A}$ is *accepted* if $Inf(\rho) \cup F \neq \emptyset$. For instance, for the ccAutomata of c_1 *subClock* c_2 with $\oint(c_1) = \oint(c_2) = Inf$ (Figure 13(a)), the acceptance criteria would be $F = \{s1\}$. The fairness between the two clocks cannot be guaranteed. A run with infinite ticks of c_2 but no tick of c_1 would be accepted, while it can not produce a valid schedule. To solve this problem, we can either expand s_1 into two sates (s_{11}, s_{12} in Figure 16) and make only s_{12} accepting, or use transition-based Büchi automata (tBA) instead, $F = \{s_1 \xrightarrow{\{c_1,c_2\},\emptyset} s_1\}$. Forcing state-based accepting sets would in fact amount to reiterating locally the general expansion from tBA to BA. But counting on transitions comes more natural.

When composing local tBAs obtained from individual constraints into a global one to represent schedules from the whole specification, we want to avoid infinite runs where some clocks are forbidden forever while non terminated. The problem is that such runs are actually accepted by the naive acceptance criterion. For this type of problems, literature has come up in the past with the notion of Generalized BA, in which a collection of acceptance conditions would be provided, each needing to be checked independently. In our case we shall associate such a condition to every specified finite or infinite clock.

Definition 1. A transition-based Generalized Büchi ccAutomata (tGBA) is a tuple $(\mathcal{A}, \oint, F)$ where $\mathcal{A}$ is a ccAutomata, $\oint$ is the finiteness specifying function, F is a finite set of elements called acceptance conditions, $F = \{c | c \in Clocks, \oint(c) = Inf\} \cup \{Term(c) | c \in Clocks, \oint(c) = Fin\}$.

A run ρ is *accepted* if transitions are labeled by each acceptance condition infinitely often. Let $Inftick(\rho) = \{c | s \xrightarrow{C,Tm} s' \in Inf(\rho), c \in C\}$ and $Infterm(\rho) = \{Term(c) | s \xrightarrow{C,Tm} s' \in Inf(\rho), Term(c) \in Tm\}$, ρ is accepted if $\forall f \in F, f \in Inftick)(\rho) \vee f \in Infterm(\rho)$. This means that in an accepted run, every infinite clock ticks infinitely often, and every finite clock terminates properly (guaranteed by the infinitely often visited termination). Under these conditions we avoid runs where clocks are forbidden to tick while they should (we avoid local deadlock of clocks) and consequently the ticking trace of an accepted run would be a valid schedule.

For Example 1, its accepting criteria would be $F = \{a, Term(b), Term(c)\}$, since both b and c are supposed to be finite while a should be infinite. Runs where at least one tick of b occurs before the third tick of a would be recognized as accepted runs, while runs that lead to state (s, t_2, r_0) would be ruled out.

Note that in practice, the number of clocks considered in defining accepting conditions could be minimized, as some of them can imply others. For instance, if $c = c_1$ `sampledOn` c_2, one only needs to find schedules in which c ticks infinitely often, as that implies c_1 and c_2 do too. Please check the detail rules for minimizing accepting conditions in Table 1.

Table 1: Acceptance condition minimizing rules

Constraint	Implication
$c = c_1$ filteredBy w, $c = c_1$ delay_n	$\oint(c_1) = Fin \Rightarrow \oint(c) = Fin$, $\oint(c) = Inf \Rightarrow \oint(c_1) = Inf$
$c = c_1$ await_n / until_n	$\oint(c_1) = Fin \Rightarrow \oint(c) = Fin$
$c = c_1$ upto c_2	$\oint(c) = Inf \Rightarrow \oint(c_1) = Inf$
$c = c_1$ concat/union/inf c_2	$\oint(c_1) = Inf \vee \oint(c_2) = Inf \Leftrightarrow \oint(c) = Inf$, $\oint(c_1) = \oint(c_2) = Fin \Leftrightarrow \oint(c) = Fin$
$c = c_1$ intersect/sampledOn c_2	$\oint(c_1) = Fin \vee \oint(c_2) = Fin \Rightarrow \oint(c) = Fin$, $\oint(c) = Inf \Rightarrow \oint(c_1) = \oint(c_2) = Inf$
$c = c_1$ sup c_2	$\oint(c_1) = Fin \vee \oint(c_2) = Fin \Leftrightarrow \oint(c) = Fin$, $\oint(c) = Inf \Leftrightarrow \oint(c_1) = \oint(c_2) = Inf$
c subClock c_1	$\oint(c) = Inf \Rightarrow \oint(c_1) = Inf$
c equality/alternate/boundedDiff_i_j c_1	$\oint(c) = \oint(c_1) = Inf$
c_1 strictPre/causes c_2	$\oint(c_2) = Inf \Rightarrow \oint(c_1) = Inf$, $\oint(c_1) = Fin \Rightarrow \oint(c_2) = Fin$

5 Schedulability checked on automata translation

Once the tGBA built, schedulability analysis is equivalent to finding the existence of accepted infinite words. This (non)emptiness check has also been identified in model-checking context (where automata are generated from temporal logical formulaes). It is straightforward to check the emptiness on plain (state) Büchi automata (existence of a loop or SCC (Strongly Connected Component) including an accepting state, of course amongst the reachable states). But it is a bit more subtle to be checked on extended tGBAs directly, and we will consider this in the current section.

In addition to checking for emptiness, we will even attempt to compute the actual subset of states from which a valid infinite word may be continued. This will then allow to trim the structure and make the automata canonical so that it accepts the same language and no useless (undesired) state remains. This is helpful if one considers an approach where the resulting automata will be used later to guide an actual run-time simulation (similar as in controller synthesis aims). Hopefully, most related model-checking techniques actually do compute such sets of states and decide later language emptiness (or not) by checking whether the initial state was preserved as one of the useful states. The previous definition of transition-based Generalized Büchi Automata can be read as: a state is *useful* if it can be the root of a path which goes infinitely often through an accepting transition for every accepting state (in our case it is a path where each infinite specified clock ticks infinitely often, and proper termination of every finite specified clock is visited infinitely often), otherwise it is *useless*. We derive an algorithm from this. It is important to note here that a state may already be useless (in the sense that it leads to no valid infinite behavior) while it takes several steps in any way before bad states where clocks get halted.

For technical reasons, we require the finiteness of state space so that the fixpoints converge (and if not finite-state, the intuitive explanations have to be adapted because an infinite path does not mean a loop or SCC anymore). If a specification contains a constraint that may cause infinite state space (`strictPre, causes, sup` and `inf`), there must a be `boundedDiff` restricting finite drifts[1]. And we want the algorithm to be easily applicable on symbolic state set representations, such as BDDs. The operations used in the algorithm (union, intersection, complementation, $Enable$, Pre), can easily be manipulated on symbolic representation of state sets. For a given specified (infinite or finite) clock c, $Enable_V(c)$ computes the set of states that immediately perform a transition where c ticks (if $\oint(c) = Inf$) or terminates (if $\oint(c) = Fin$). $Pre_V(Z)$

[1] Actually this restriction is overstrict, because of the interactions among constraints, the state space of a specification may be finite even if it has unbounded constraints. However, since the semantics is built constructively, it is not easy to check the finiteness from the syntactic constructors directly. So right now, we restrict to bounded CCSL [2] instead of finiteness checking

computes the set of predecessors of states in Z. Subscript V is the scope where computations are carried.

$$Enable_V(c) = \begin{cases} \{s \in V | \exists s^{'} \in V, s \xrightarrow{C,Tm} s^{'}, c \in C\}, & \oint(c) = Inf, \\ \{s \in V | \exists s^{'} \in V, s \xrightarrow{C,Tm} s^{'}, Term(c) \in Tm\}, & \oint(c) = Fin \end{cases}$$

$Pre_V(Z) = \{s^{'} \in V | \exists s \in Z, s^{'} \xrightarrow{C,Tm} s\}$.

Our algorithm (Algorithm 1) consists of three steps:

The first step computes the set of *potentially useful* states, PU. A state is *potentially useful* if in its future, every infinite clock can tick and every finite clock can properly terminate (may several times, infinitely, or immediately). For each specified clock, we compute independently the set of states that may reach a transition where it ticks (infinite specified) or terminates (finite specified) in a finite but unbounded number of steps, which is a straightforward backward (smallest) fixpoint of functional $f(X) = Enable_S(c) \cup Pre_S(X)$. PU is then the intersection of all those result sets. PU is closed by co-reachability (if a state is in it, all states that may reach it also are). So if PU does not contain an initial state then we are done, the CCSL specification is not schedulable.

```
1  algorithm Trimming
   Input: tgba = ((S, Clocks, Terms, I, T), ∮, F)
   Output: set of states U
2  begin
3      set of states PU,OU,U,Ufront,Uprev;
4      if S=∅ then return;
5      PU :=markState(tgba,S);
6      OU :=markState(tgba,PU);
7      U :=OU; Ufront := OU;
8      while U≠ Uprev do
9          Uprev:=U;
10         Ufront:=Pre_S(Ufront);
11         U:=U ∩ Ufront;
12     end
13     if U=∅ then tgba is not schedulable
14 end
15 algorithm markState
   Input: tgba = ((S, Clocks, Terms, I, T), ∮, F), scope state set SS ⊆ S
   Output: set of states intersect
16 begin
17     set of states cLabeled,cLabeledfront;
18     intersect := S;
19     for every specified clock c ∈ Clocks do
20         cLabeled :=Enable_SS(c); cLabeledfront :=Enable_SS(c);
21         while cLabeledfront≠ ∅ do
22             cLabeledfront:=Pre_SS(cLabeledfront)\cLabeled;
23             cLabeled:=cLabeled ∪ cLabeledfront;
24         end
25         intersect := intersect ∩ cLabeled;
26     end
27 end
```

Algorithm 1: The Trimming algorithm

Obviously a state that is not potentially useful is useless (from it there are clocks that get halted). But a PU state may not turn out to be actually useful: consider the case of a loop (or more generally a SCC) inside which two distinct clocks never tick, but from which there may be two actual transitions (leaving the SCC at different points), each of them allowing one of the two clocks to tick (and never the other

anymore). Then seemingly all states inside the SCC were labeled by both (here, all) clocks, while in fact the tickings cannot occur in the same path. (Case for finite clocks on proper terminations is similar.) Our solution to this is to apply the same algorithm as before, but this time restricting to PU as the set of reachable states (as thus regarding accepting transitions only if both their target and source states are in it). The set OU of once-useful states contains states that remain potentially useful. OU is also co-reachability closed.

The last step is to check the emptiness and get the set of useful states, U. We check if there is at least one loop (or SCC) inside U, by computing a greatest fixpoint on function $f(X) = OU \cap Pre_S(X)$. It computes the set of states from which there is an unbounded path inside U, containing SCCs plus extra states that may lead to them (the emptiness problem is therefore preserved between SCCs and this slight extension). U is the set of states preserved by this step. Since the state space is finite, the algorithm eventually ends.

Proposition 5.1 *A state is in U iff there is an infinite path from it in which each infinite clock ticks infinitely often and the termination of each finite clock is visited infinitely often (or in general terms where an accepting transition is performed infinitely often for each acceptance condition of the tGBA).*

Proof (sketch): The $\Leftarrow$ direction is easy: if there is such an infinite path, then the states found in it are all in OU by definition, and fold itself into a SCC in the end, so that all states remain in U.

For the $\Rightarrow$ part: if s is in U, then it is in OU. From s, for a given infinite (resp. finite) specified clock c, (or equivalently, an accepting condition in generalized Buchi mode), there has to be a transition labeled as "c ticks" (resp. "c terminates"), with source and destination states (t, u) in OU. Now these states can also be chosen in U, otherwise the last state s' in the path from s would not remain labeled by at least one of the clock during the second step, a contradiction. Now we can reiterate this process from state u and any clock since U ultimately ends in (one or several) SCC, and progressively build an accepting infinite path. This concludes the proof. □

6 Conclusion and future work

In this paper we use a translation from logical time constraint language to automata expansion to study schedulability. This work bears some similarity with previous attempts by Alur and Weiss [3, 4], which define schedules as infinite words expressed in regular expressions and then construct corresponding Büchi automatas, and have counterparts in more classical real-time and timed automata in[5, 6, 7, 8]. Comparing to them, we use CCSL as the bridge to connect scheduling and automata theories. This allows to take full of advantages of CCSL in modeling, such as the flexibility provided by logical time, multi-form time bases, unified expressing of both functional requirements and extra performance constraints. And we can construct automata naturally from CCSL specifications, avoiding the high costing of translation from regular expressions to automata.

We propose an algorithm for emptiness-checking and canonical trimming of translated automata, tGBA. As far as we know, it is a new way of handling generalized acceptance conditions, comparing to the existing two classes of emptiness-check algorithms: nested depth-first searches (NDFSs) based [9, 10, 11], and strongly connected components (SCCs) computation based [12, 13, 14]. And we believe it has relevance larger than our specific problem, being extended for general model-checking for transition or state GBAs.

In the future we want to inspect how finer relations between logical clocks can be used to detect unschedulability at construction time, in states even prior the full model is built. And we also want to make use of "latency-insensitive" property embedded in the interactions among constraints, which is distinct valid schedules depart from one another by shifting the exact timing placement they each assign to otherwise independent clock ticks.

References

[1] Charles André. Syntax and semantics of the clock constraint specification language (ccsl). Research Report RR-6925, INRIA, 2009.

[2] Régis Gascon, Frédéric Mallet, and Julien Deantoni. Logical time and temporal logics: comparing UML MARTE/CCSL and PSL. In *18th International Symposium on Temporal Representation and Reasoning (TIME'11)*, pages 141–148, Lubeck, Germany, September 2011.

[3] Rajeev Alur and Gera Weiss. Regular specifications of resource requirements for embedded control software. In *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '08, pages 159–168, Washington DC, USA, 2008. IEEE Computer Society.

[4] Rajeev Alur and Gera Weiss. Rtcomposer:a framework for real-time components with scheduling interfaces. In *Proceedings of the 8th ACM international conference on Embedded software*, EMSOFT '08, pages 159–168, New York, NY, USA, 2008. ACM.

[5] Yasmina Abdeddaim and Damien Masson. Real-time scheduling of energy harvesting embedded systems with timed automata. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), IEEE 18th International Conference on*, pages 31 –40, 2012.

[6] Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen. Optimal scheduling using priced timed automata. *SIGMETRICS Perform. Eval. Rev.*, 32(4):34–40, 2005.

[7] Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Schedulability analysis of fixed-priority systems using timed automata. *Theoretical Computer Science*, 354(2):301 – 317, 2006.

[8] Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149 – 1172, 2007.

[9] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 233–242. Springer Berlin / Heidelberg, 1991.

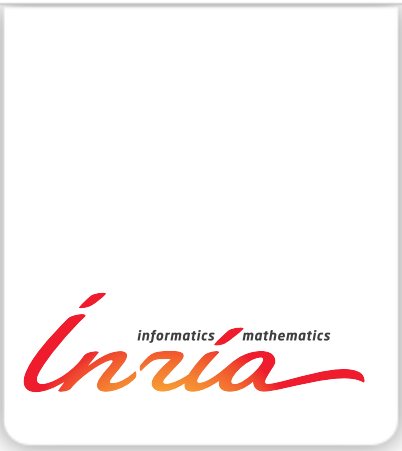
[10] H. Tauriainen. Nested emptiness search for generalized buchi automata. *Fundam. Inf.*, 70(1):127–154, 2005.

[11] Heikki Tauriainen. On translating linear temporal logic into alternating and nondeterministic automata. Research Report A83, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, December 2003.

[12] Jaco Geldenhuys and Antti Valmari. More efficient on-the-fly ltl verification with tarjan's algorithm. *Theor. Comput. Sci.*, 345(1):60–82, November 2005.

[13] Hammer Moritz, Alexander Knapp, and Stephan Merz. Truly on-the-fly ltl model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 191–205. Springer Berlin/Heidelberg, 2005.

[14] Jean-Michel Couvreur, Alexandre Duret-Lutz, and Denis Poitrenaud. On-the-fly emptiness checks for generalized büchi automata. In *Model Checking Software*, volume 3639 of *Lecture Notes in Computer Science*, pages 903–903. Springer Berlin / Heidelberg, 2005.



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399